LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-PROC-666300

# Strong Scaling Bottleneck Identification and Mitigation in Ares

I. Karlin, M. Collette

January 16, 2015

Strong Scaling Bottleneck Identification and Mitigation in Ares
Los Alamos, NM, United States
October 20, 2014 through October 24, 2014

## Disclaimer

# (U) Strong Scaling Bottleneck Identification and Mitigation in Ares

*Ian Karlin, Mike Collette*

karlin1@llnl.gov, phone (925) 422-0179
Lawrence Livermore National Laboratory, Livermore, CA
Topical Areas: (Computer Science, 20.1 Advanced / Heterogeneous / Exascale Architecture, 20.2 Parallelism / Concurrency, 20.3 Performance / Scaling / Optimization)

## Abstract

(U) With processor clock speeds staying steady or decreasing across HPC systems, reducing time to solution of a resolved problem requires strong scaling. Decreasing memory capacity per core on current ASC advanced architecture machines is also forcing strong scaling of problems, if a programmer wants to utilize all compute resources. In addition, future computer systems are expected to contain two-level memory systems, with approximately an order of magnitude difference in size and performance between levels. Codes that can strong scale to fit fully, or mostly, within the smaller faster memory will perform significantly better than those that cannot. In this paper, we present a strong scaling study of Ares running the 2D Sedov problem on BG/Q. The study indicates that physics sections of Ares already strong scale well down to as few as 64 zones per processor. However, as problem size per core decreases runtime libraries (MPI and malloc) and user support features dominate runtime. While some of these performance bottlenecks require changes in the user workflow model to increase performance, we are able to show that by linking with different runtime libraries, significant performance gains occur for both our test and production problems.

## Introduction

Time to solution is important for users of many scientific computing applications, as varied as weather prediction to NIF experimental design [1]. Historically, time to solution has decreased as processor clock frequencies have increased. However, power efficiency concerns, and an end of Denard scaling have resulted in stagnating or decreasing clock frequencies recently [6]. Therefore, to decrease time to solution for a given resolution problem, strong scaling is required.

While time to solution is an important reason to strong scale a problem, capacity and speed of the memory system are another reason to pursue strong scaling a problem. The cost of main-memory is not decreasing as fast as the cost of processing, resulting in computer systems that have with less memory capacity per core. That effect is amplified by the use of slower simpler cores that often rely on hardware threads rather than out of order execution to hide latency. The BlueGene/Q

architecture, Xeon Phi and GPUs are just some examples of architecture designs that trade the speed and latency hiding capability of a single fat core for multiple slower, simpler and multi-threaded cores.

Finally, on future machines in order to achieve both the bandwidth and capacity goals required to effectively run large scale DOE calculations, it is expected that multiple levels of main-memory will be present. The multiple levels of memory will likely include a fast in-package stacked memory that is situated on the same silicon interposer as the compute chip, and a second memory that is about an order of magnitude larger and slower than main memory. While the entire memory of a machine will be needed for high resolution calculations, problems that do not need all of memory will complete significantly faster if they can fit, or nearly fit, within the fast in-package memory. However, in order to run efficiently from the fast memory, they will need to strong scale effectively.

In this paper, we show the challenges of efficiently strong scaling the Ares application. We provide data that shows how Ares scaled before our modifications. We then describe runtime library changes that improved the performance of both the Sedov problem and production problems. Finally, we present conclusions and future work, where we also identify other usage changes, programming improvements and hardware features that could help more efficiently strong scale Ares.

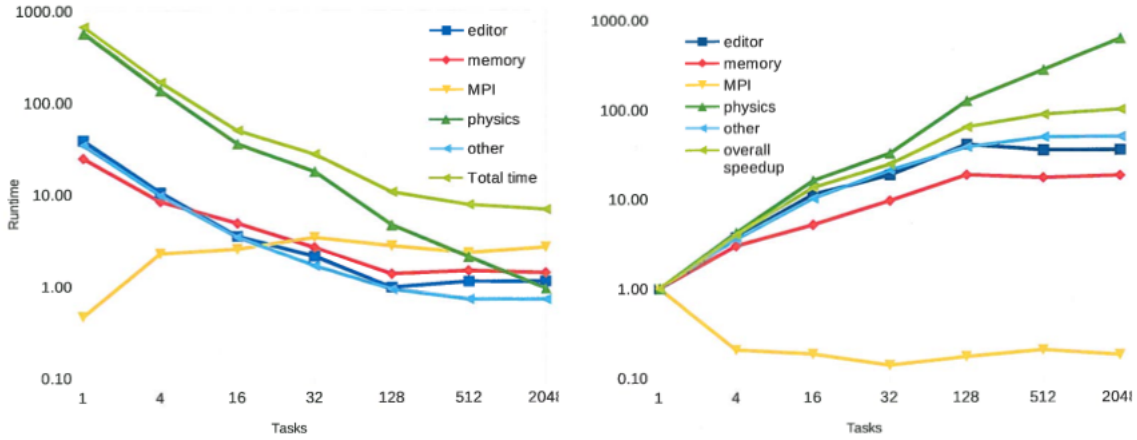
## Experimental Setup and Results

Our strong scaling study of Ares used a 2D Sedov problem on a cartesian grid. The experiments were run on a BG/Q system using one MPI task per core. Our base problem was run on a single processor using 131,072 zones and was strong scaled to 2048 processors with each core running an MPI task having 64 zones. All runtime profiling data was collected using the performance monitoring tool HPCToolkit [2] from Rice University. All message passing data, including message sizes and the amount of time spent sending messages was collected using mpiP [3].

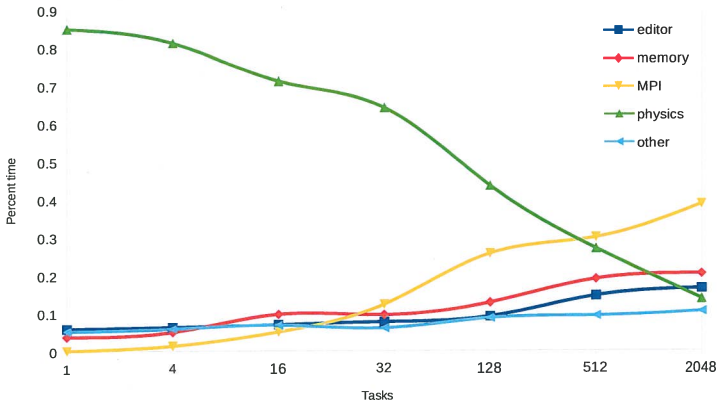
Figure 1 shows the runtime of the Sedov problem in seconds in the left plot and the scaling of the problem on the right graph as task count increases. The lower chart shows the percentage of the runtime spent in various code sections. On top two charts charts there is an overall program time and on all three there are lines for five sub-components discussed below. The **editor**, allows users to adjust the physics model being used based on a physical change in a zone, such as temperature increasing above a threshold. **Memory** measures all the time spent in functions that are dedicated to allocating and freeing data structures within a timestep loop. While not needed for the Sedov problem these features help reduce the overall memory footprint of more complex problems.

**MPI** measures the amount of time spent in the functions that call MPI routines. The MPI time includes packing and unpacking of data as well as the time spent in the MPI library itself. **Physics** is a measure of the amount of time spent in the physical simulation loops. **Other** captures all the code not in another region. This includes time spent in code that could be characterized multiple ways, that took a small percentage of the runtime, and Ares internal timers that monitor performance during production runs. We worked to identify enough time spent in named functionality so that other was always less than 10% of the runtime.

From Figure 1 we observe that only the time spent in the physics code continues to decrease as at




(a) Runtime (b) Scaling




(c) Percentage Breakdown

**Figure 1: Runtime and scaling of 2D Sedov problem on BG/Q.**

| Tasks | Min Task MPI % | Average MPI % | Max Task MPI % | Largest Message | Average Message Size |
|---|---|---|---|---|---|
| 16 | 2.48 | 3.55 | 6.06 | 5152 Bytes | 1558 Bytes |
| 128 | 8.62 | 11.00 | 18.97 | 1312 Bytes | 425 Bytes |
| 512 | 14.20 | 16.98 | 30.27 | 672 Bytes | 219 Bytes |
| 2048 | 18.60 | 21.98 | 37.19 | 352 Bytes | 120 Bytes |

**Table I: Time spent in MPI and message size characteristics.**

all processor counts with near perfect scaling 123x up to 128 processors. Beyond 128 processors the physics still scales well with over a 2x improvement when core count is increased by 4x. While accounting for over 80% of the runtime on a single core, by 2048 cores the physics code consumes less time than each of the editor, MPI and memory sections. The MPI section starts as the least expensive part on one to sixteen cores, but increases to the second most expensive at 32 and most expensive runtime component at 512 and larger. MPI is the only component that does not decrease in cost as processor count increases.

The other three components (memory, editor and other) all show a runtime decrease as processor count increases. However, their improvements are significantly smaller than the physics runtime reductions and no component has a runtime reduction beyond 512 processors. Note that the largest speedup of 48x occurs from the other section and all scale at less than 33% efficiency at 128 processors. The poor scaling of these three components and MPI are what contribute to the overall speedup only reaching 98x at 2048 cores. However, the time spent in physics is sufficiently large that runtime scaling efficiency is over 75% on 32 processors and just under 50% on 128.

Using mpiP we also separately analyzed load balance, time in the MPI library and message size. A summary of the MPI data is presented in Table I. We observe that even for 16 tasks contained within a single node there is significant imbalance. In addition, the largest messages are about 5 KB for 16 tasks and that shrinks significantly for large task counts. This is significantly smaller than 10 KB where IBM applications experts say throughput matters more than latency. Data from mpiP backs this up, showing that the average Isend at 2048 tasks is 5.7 $\mu s$ while for 16 tasks the average Isend takes 8.3 $\mu s$. Therefore, MPI library overhead and communication latency are a larger cost than data motion for this problem.

## Improving Performance Through Faster Runtimes

The data we collected from this experiment led us to investigate two runtime library changes to improve Ares performance. On the BlueGene/Q system there are 6 versions of MPI with various levels of feature support and argument correctness checking at runtime. Each level of support adds additional overhead in the MPI library that delays the time it takes to send a message. Internal testing at Lawrence Livermore has shown that for small messages, end to end message latency can improve by up to 3x by using a “fast MPI” that does not check arguments or support newer features such as thread safe message passing. By switching to this “fast” version, we saw a 4% performance improvement for our test problem at the higher processor counts. Runtime gains were constant regardless of problem size, showing that fast MPI libraries are more important when messages are a larger part of the runtime and most messages are small.

Memory allocation and deallocation took over 10%, and eventually rising to 20%, of runtime at all

processor counts 16 and larger. The default malloc library on most systems is optimized for large allocations. It also will usually aggressively give freed memory back to the operating system. However, when a problem is strong scaled, most allocations are smaller, with data more frequently allocated and freed. Specialized malloc libraries are sometimes better tuned for smaller more frequent allocations. One example is google's tcmalloc [4], which locally caches freed data and keeps a small pool of data available. By having mallocs handled by the operating system less regularly, tcmalloc can increase the performance of small allocations. Previous resulst using LULESH [5] have shown it can be effective at reducing the cost of mallocs in hydrodynamics codes and is an example of lessons learned in a proxy application making their way back into production applications. We did not try tcmalloc on the Sedov problem, however it resulted in up to a 10% runtime improvement on production problems, and when combined with the "fast" MPI we saw runtime gains of 10-15% in production.

## Conclusions and Future Work

In this paper we presented a strong scaling study of the 2D Sedov problem on BlueGene/Q using the Ares code. We showed that the physics code in Ares scales extremely well, but that other components including the MPI, memory management and editor features do not, for latency reasons. However, by using faster runtime libraries, we were able to reduce the cost of memory management and MPI with a resulting 10-15% performance gain for production problems.

In this paper we did not address code changes that might have further helped strong scalability, how future hardware might improve it, or how threading runtimes might impact strong scalability. In future work we plan to explore how improving the code base helps scalability. Altering user workflow behavior would help as well, by for example, calling the editor less frequently. Future hardware with heterogenous processing capabilities, may improve scalability, by allowing us to run physics code on throughput optimized cores, and runtimes on latency optimized cores. However, multi-level memory systems and less memory per core both present performance challenges and opportunities. Finally, we are currently investigating how OpenMP runtime performance effects application performance.

## Acknowledgements

The authors would like to thank Bob Walkup from IBM for porting tcmalloc to the BlueGene Q machines at LLNL. We also thank John Gyllenhall for support of the "fast" MPI libraries. This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.0 (LLNL-PROC-666300)

## References

[1] Arthur A. Mirin, David F. Richards, James N. Glosli, Erik W. Draeger, Bor Chan, Jean-luc Fattebert, William D. Krauss and Tomas Oppelstrup Toward Real-Time Modeling of Human Heart Ventricles at Cellular Resolution: Simulation of Drug-Induced Arrhythmias SC '12 Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis. IEEE Computer Society Press, Los Alamitos, CA, USA, 2012, Article

2, 11 pages.

[2] Laksono Adhianto, Sinchan Banerjee, Mike Fagan, Mark Krentel, Gabriel Marin, John Mellor-Crummey, and Nathan R. Tallent. HPCToolkit: Tools for performance analysis of optimized parallel programs. Concurrency and Computation: Practice and Experience, 22(6):685-701, 2010.

[3] Vetter, J.S. and M.O. McCracken Statistical Scalability Analysis of Communication Operations in Distributed Applications. Proc. ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming (PPOPP), 2001.

[4] Sanjay Ghemawat. TCMalloc: Thread-Caching Malloc. 2014, http://google-perftools.googlecode.com/svn/trunk/doc/tcmalloc.html.

[5] Ian Karlin, Jim McGraw, Jeff Keasler and Bert Still. Tuning the LULESH Mini-app for Current and Future Hardware NECDC 2012, Lawrence Livermore National Laboratory (LLNL), Livermore, CA.

[6] John Shalf, Sudip Dosanjh, and John Morrison. Exascale computing technology challenges. High Performance Computing for Computational Science-VECPAR 2010 (2011): 1-25.